Available online at www.sciencedirect.com

SciVerse ScienceDirect

North American Search Based Software Engineering Symposium 00 (2015) 1–15

NASBASE

www.elsevier.com/locate/procedia

A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns

Ali Ouni[a,*], Marouane Kessentini[b], Houari Sahraoui[c], Mel Ó Cinnéide[d], Kalyanmoy Deb[e], Katsuro Inoue[a]

[a] *Graduate School of Information Science and Technology, Osaka University, Japan*
[b] *University of Michigan, USA*
[c] *DIRO, Université de Montréal, Montréal, Canada*
[d] *University College Dublin, Ireland*
[e] *Michigan State University, USA*

Abstract

Refactoring is widely recognized as a crucial technique applied when evolving object-oriented software systems. Refactoring has promised, if applied well, to improve software readability, maintainability and extendibility. In general, to improve software quality, most of existing studies focus on the correction of anti-patterns or enhancing specific quality metrics. However, this may not be sufficient to make the source code easier to understand and modify. The introduction of design patterns that represent good design practices represents an efficient way improve the quality of systems; but very few works consider the use of refactoring to introduce design patterns. We propose, in this paper, an automated multi-objective refactoring recommendation approach to (1) improve design quality (as defined by software quality metrics), (2) fix anti-patterns, and (3) introduce design patterns. To evaluate our approach, we conducted a quantitative and qualitative evaluation using a benchmark composed of four open source systems. The statistical analysis of the results confirms the efficiency of our proposal compared to the state-of-the-art of refactoring approaches.

Keywords: Search-based software engineering, Refactoring, Code-smells, Anti-patterns

1. Introduction

As software systems change and evolve continuously, there is a constant need for high-quality software. To improve the quality of software systems, one of the widely used techniques is *refactoring* - the process of improving software design structure without changing its external behavior. Refactoring can help software developers to reduce the time required for adding new requirements, correcting bugs, understanding the existing implementation, and modifying the code to improve its quality. Consequently, various refactoring tools have been proposed [8] [10] [11] [12].

* Corresponding author. Tel.: +0-000-000-0000 ; fax: +0-000-000-0000 .
E-mail address: ouniali@iro.umontreal.ca

Despite its significant benefits, recent studies showed that automated refactoring tools are underused most of the time [4] [5] [8]. One of the possible reasons is that most existing refactoring tools [22] [24] [11] focus mainly only on improving some quality metrics (e.g., coupling, cohesion, complexity, etc.). For instance, improving software quality factors does not mean that *anti-patterns* [15] or bad-design practices that may exist are fixed. Thus, quality metric values can be significantly improved but the original program may still contain a considerable number of anti-patterns, which may lead to maintenance and evolution difficulties. On the other hand, *design patterns* are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to change [25]. Design patterns can be automatically introduced using refactoring [1] [20], however, most existing refactoring tools do not consider the use of design patterns to fix anti-patterns and improve the quality of software systems. In addition, applying a design pattern where it is not needed is highly undesirable as it introduces an unnecessary complexity to the system for no benefit [28].

To address the above-mentioned challenges, we propose a novel approach to guide the introduction of design patterns by fixing anti-patterns and improving the overall quality of the system while avoiding the introduction of semantic incoherencies to the design [27]. To the best of our knowledge, this is the first work that suggests refactoring strategies to combine introducing design patterns and fixing anti-patterns to improve software design quality within one framework. To this end, we have developed a multi-objective optimisation approach supported by a tool called MORE (Multi-Objective Refactoring REcommendation) to find the best compromise between 1) improving software quality, 2) fixing anti-patterns and 3) introducing design patterns while satisfying a set of constraints to ensure the semantic coherence of the refactored program. More specifically, the primary contributions of the paper are as follows:

1. We introduce a multi-objective search-based refactoring approach to improve software quality attributes (i.e., flexibility, maintainability, etc.), introduce "good" design practices (i.e., design patterns) and fix "bad" design practices (i.e., anti-patterns). We implemented our approach in a tool called MORE. We present a set of constraints, for each refactoring operation, in order to ensure the semantic coherence of the refactored program, e.g., that a method is not moved to a class where it makes no sense.
2. We present an empirical study based on a quantitative and qualitative evaluation using a benchmark composed of four real-world software projects of various sizes. The quantitative evaluation investigates the efficiency of our approach in fixing four types of anti-patterns (God class, Feature Envy, Data Class, and Spaghetti Code), introducing three types of design patterns (Factory Method, Visitor, and Singleton), and improving six quality attributes according to the popular software quality model QMOOD [26]. For the qualitative evaluation, we manually evaluate the usefulness of our approach in terms of refactoring feasibility to assess whether our refactorings are meaningful and coherent with the program semantics.

The remainder of this paper is organised as follows. Section 2 presents the background and related work. Section 3 introduces our search-based approach, MORE. Section 4 describes the design of the empirical study to evaluate our approach, and presents the experimental results. Finally, in Section 5, we conclude and describe our future research directions.

2. Background

2.1. Definitions

Refactoring is defined as the process of changing the structure of software while preserving its external behavior. The term refactoring was introduced by Opdyke and Johnson [14], and popularized by Martin Fowler's book [2]. The idea is to reorganize variables, classes and methods mainly to facilitate future adaptations and extensions. This reorganization aims at improving different aspects of software quality such as maintainability, extensibility, reusability, etc. [3].

Refactoring is known to be an effective way to fix anti-patterns. Anti-patterns, also called code-smells or design defects, are symptoms of poor design and implementation practices that describe a bad solution to a recurring design problem that leads to negative effects on code quality [15]. Software engineers often introduce anti-patterns unintentionally during the initial design or during software development due to bad design decisions, ignorance or time pressure. Consequently, anti-patterns that cause problems should be removed from the software design as early as possible. In this paper, we focus on the following four design anti-pattern types in evaluating our approach: God Class, Feature Envy, Data Class, and Spaghetti Code [2] [15]. We choose these anti-pattern types in our experiments because they are the most important and frequently-occurring ones in industrial projects based on recent studies [10] [19] [29]. Moreover, it is widely believed that anti-patterns have a negative impact on software quality and often lead to bugs and failures [29] [30].

On the other hand, design patterns are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to change [25]. We focus in this paper on a subset of the Gamma et al. design patterns, including Factory Method, Visitor and Singleton [25]. We choose these design patterns because they address problems related to classes and their associations to improve the structure of an object-oriented software design.

2.2. Related work

Search-based Software Engineering (SBSE) has been used successfully to automate many software engineering tasks [31], including the problem of automated refactoring to improve some aspect of the software [10] [11] [12] [20] [22] [28] [35]. These approaches cast refactoring as an optimization problem using different optimisation techniques such as hill climbing, genetic algorithms, simulated annealing, etc. Existing automated refactoring approaches can be classified into three main categories depending on the goal: 1) to improve quality factors; 2) to fix anti-patterns; and 3) to introduce design patterns.

Improve design quality. The majority of existing search-based refactoring approaches use software metrics as objective function(s) to find a good sequence of refactorings. Seng et al. [22] have proposed a single-objective optimization approach that uses a genetic algorithm to suggest a list of refactorings to improve software quality. The search process employs a single fitness function to maximize a weighted sum of several quality metrics (coupling, cohesion, complexity and stability). Similarly, O'Keeffe and Ó Cinnéide [12] used different local search-based techniques such as hill climbing and simulated annealing to find a good sequence of refactorings to apply. They used a fitness function comprising eleven weighted object-oriented design metrics from the QMOOD metrics suite [26]. Fatiregun et al. [32] showed how search-based transformations could be used to reduce code size and construct amorphous program slices. However, they use small atomic level transformations, rather than refactorings. In addition their aim was to reduce program size rather than to improve its structure/quality. Harman and Tratt [11] proposed the first search-based approach that uses Pareto optimality to combine two quality metrics, in this case CBO (coupling between objects) and SDMPC (standard deviation of methods per class). Their aim is to find a sequence of "move method" refactorings to find the best trade-off between these two metrics. Although these approaches are powerful enough to improve quality as expressed by software quality metrics, this improvement does not mean that they are successful in removing actual instances of anti-patterns.

Fixing anti-patterns. Search-based refactoring has been used to correct anti-patterns by several authors. Kessentini et al. [34] proposed an approach using a mono-objective genetic algorithm to find a sequence of refactorings that attempts to minimize the number of anti-patterns detected in the source code. Ouni et al. [27] proposed a multi-objective formulation of refactoring to find the best compromise between fixing code-smells, and semantic coherence using two heuristics related to vocabulary similarity and structural coupling. The idea

behind these two heuristics is to avoid semantic coherence violation when moving methods/fields between classes.

Introduce design patterns. One of the earliest works in automated introduction of design patterns was that of Ó Cinnéide and Nixon [1] [28] who presented a methodology for the development of design pattern transformations in a behavior preserving fashion. They identified a number of "pattern aware" composite refactorings called mini-transformations that, when composed, can create instances of design patterns. They defined a starting point for each pattern transformation, termed a precursor. This is where the basic intent of the pattern is present in the code, but not in its most flexible pattern form. More recently, Jensen and Cheng [20] have proposed the first search-based refactoring approach that supports composition of design changes and makes the introduction of design patterns a primary goal of the refactoring process. They used genetic programming, software metrics, and the set of mini-transformations identified by Ó Cinnéide and Nixon [28] to identify the most suitable set of mini-transformations to maximize the number of design patterns in a software design. Recently, Ajouli et al. [18] have described how to use refactoring tools (IntelliJ, and Eclipse) to transform a Java program conforming to the Composite design pattern into a program conforming to the Visitor design pattern with the same external behavior, and vice versa.

Furthermore, researchers have proposed various ways to improve automated refactoring. For instance, Murphy-Hill et al. [7] [8] [45] proposed several techniques and empirical studies to support refactoring activities. In [45] [46], the authors proposed new tools to assist software engineers in applying refactoring manually such as selection assistant, box view, and refactoring annotation based on structural information and program analysis techniques. Recently, Ge and Murphy-Hill [16] have proposed new refactoring tool called GhostFactor that allow the developer to transform code manually, but check the correctness of her transformation automatically. However, the correction is based mainly on the structure of the code and do not consider its semantics. Mens et al. formalize refactoring by using graph transformations [43]. Bavota et al. [44] automatically identify method chains and refactor them to cohesive classes using extract class refactoring. The aim of these approaches is to provide specific refactoring strategies; the aim of our approach is to provide a generic and automated refactoring framework to help developers to refactor their code.

3. Approach: MORE

This section describes the principles that underlie the proposed approach, called MORE (Multi-Objective Refactoring REcommendation) for improving software quality, fixing anti-patterns, and introducing design patterns while maintaining the coherence of the refactored code. We first describe our approach, its components and the semantic constraints employed. Then, we provide a detailed description of the search algorithm, NSGA-II [17], and its adaptation for the refactoring suggestion problem.

3.1. Approach overview

The general structure of MORE is described in Fig. 1. It takes as input the source code of the program to be refactored, and as output it produces a sequence of refactorings that find the optimal trade-off between: 1) improving quality, 2) fixing anti-patterns, and 3) introducing design patterns. MORE comprises seven components that will be described in the following paragraphs.

Source code parser and analyzer (label A). This component aims at parsing and analyzing the source code of the program being refactored. Our approach is based on Soot [37], a Java optimization framework. The original source code is analyzed in order to extract from it the relevant code elements (i.e., classes, methods, attributes, etc.) and the existing relationships between them. The outputs are 1) the parsed code in a specific representation that is simple to manipulate during the search process, and 2) a call graph for the entire program that will be used

for calculating semantic constraints and software metrics (e.g. coupling, cohesion, etc.).

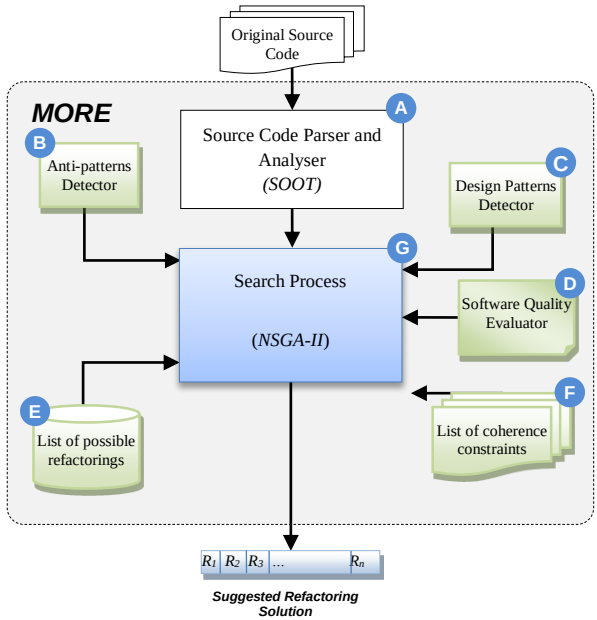

Fig. 1. Architecture of MORE

Anti-pattern detector (label B). This component scans the entire software program in order to find existing anti-pattern instances using a set of anti-pattern detection rules [10]. Detection rules are expressed in terms of metrics and threshold values. Each rule detects a specific anti-pattern type (e.g., *blob*, *feature envy*, etc.) and is expressed as a logical combination of a set of quality metrics/threshold values. These detection rules are generated from real instances of code-smells using genetic algorithm [10]. When executed, the anti-patterns detector returns a list of existing anti-pattern instances in the current version of the software. For more details about the anti-patterns detector, the interested reader is invited to confer to [10].

Design pattern detector (label C). This component is responsible for detecting existing design pattern instances in the code being refactored. Extensive research has been devoted to develop techniques to automatically detect instances of design patterns in the code and design levels. In our approach, we are using a detection mechanism that is inspired by the work of Heuzeroth et al. [38]. A design pattern *P* is defined by a tuple of program elements such as classes, methods conforming to the restrictions or rules of a certain design pattern. The detection strategy [38] is based on static and dynamic specifications of the pattern. In MORE, we use only the static specifications with a post-processing step to eliminate redundancies. Static specifications are based on predicates to identify the types of code elements like classes, methods, calls, etc. and relate them to the roles in the pattern. Each design pattern $P=(S_c, S_r)$ is then identified as a tuple of code elements S_c that are components of *P*, and a set of binary relations S_r between them. For instance, according to its specifications, the Factory method pattern is defined as follows: $P_{FactoryMethod}=(S_c, S_r)$ where

- S_c={*AbstractCreator*, *ConcreteCreator*, *ProductInterface*, *ConcreteProduct*, *FactoryMethod*, *ConcreteFactoryMethod*} represents the code elements involved in the design pattern.
- S_r represents the minimum set of binary relations between the elements of S_c that should be satisfied for the current design pattern, i.e.,

{ConcreteCreator inherits from AbstractCreator,
ConcreteProduct implements ProductInterface,
AbstractCreator defines FactoryMethod,
FactoryMethod returns ProductInterface,
ConcreteCreator defines ConcreteFactoryMethod,
ConcreteFactoryMethod returns ConcreteProduct,
ConcreteFactoryMethod overrides FactoryMethod}

Software quality evaluator (label D). This component consists of a set of software metrics that serves to evaluate the software design improvements after refactoring. Hence, the expected benefit from refactoring is to enhance the overall software design quality as well as fixing anti-patterns [2]. We use, in our approach the QMOOD (Quality Model for Object-Oriented Design) model [26] to estimate the effect of the suggested refactoring solutions on quality attributes.

List of refactorings (label E). The MORE approach currently supports the following refactoring types: Move method, Move field, Pull up field, Pull up method, Push down field, Push down method, Inline class, Extract method, Extract class, Move class, Extract superclass, Extract subclass, and Extract interface [2]. We selected these refactoring because they are the most frequently used and they are implemented in modern IDEs such as Eclipse and Netbeans.

We also considered specific blocks of refactorings to automatically introduce different types of design pattern instances. We are referring to some guidelines given in the literature for introducing instances of design patterns [18] [13]. MORE currently supports the following three design pattern types: Visitor, Factory Method, and Singleton. We selected these three design patterns because they are frequently used in practice, and it is widely believed that they embody good design practice [25]. The algorithms here apply a typical implementation of the pattern, and leave to the developer the task of tailoring the implementation to fit the context, if necessary. Note that if an atomic refactoring fails due to a non-satisfied precondition, the whole refactoring sequence that applies the design pattern will be rejected by MORE.

Coherence constraints checker (label F). The aim of this component is to prevent incoherent changes to code elements. Most refactorings are relatively simple to implement and it is straightforward to show that they preserve behaviour assuming their pre-conditions are true [3]. However, until now there is no consensual way to investigate whether a refactoring operation is semantically feasible and meaningful [27]. Preserving behavior does not mean that the coherence of the refactored program is also preserved. For instance, a refactoring solution might move a method `calculateSalary()` from the class `Employee` to the class `Car`. This refactoring could improve program structure by reducing the complexity and coupling of the class `Employee` while preserving program behavior. However, having a method `calculateSalary()` in the class `Car` does not make sense from the domain semantics standpoint. To avoid this kind of problem, we use a set of semantic coherence constraints that must be satisfied before applying a refactoring in order to prevent incoherent changes to code elements [48].

Search process (label G). Our approach is based on a multi-objective optimization using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [17] to formulate the refactoring suggestion problem. We selected NSGA-II because it is widely-used in the field of multi-objective optimization, and demonstrates good performance compared to other existing metaheuristics in solving many software engineering problems [31]. Thus our approach can be classified as Search Based Software Engineering (SBSE) [17] for which it is established best practice to define a representation, fitness functions and computational search algorithm [31]. Referring to Figure 1, the search process takes as input the source code that is then parsed into a more manipulable representation (label A), a set of anti-pattern detectors (label B), a set of design patterns detectors (label C), a software quality evaluator (label D) that evaluates post- refactoring software quality, a set possible

refactoring operations to be applied (label E), and set of constraints (label F) to ensure semantic coherence of the code after refactoring. As output, our approach suggests a list of refactoring operations that should be applied in the right order to find the best compromise between fixing anti-patterns, introducing design patterns, and improving design quality.

3.2. Semantic constraints

Unlike existing automated refactoring approaches, MORE defines and uses a set of semantic constraints to prevent arbitrary changes that may affect the semantic coherence of the refactored program. Hence, applying a refactoring where it is not needed is highly undesirable as it may introduce semantic incoherence and unnecessary complexity to the original design. To this end, we considered several semantic constraints defined in our previous work [27] [48]: Vocabulary-based similarity constraint (VS), Dependency-based similarity constraint (DS), Implementation-based similarity constraint (IS), Feature inheritance usefulness constraint (FIU), and Cohesion-based dependency constraint (CD).

We introduced some semantic constraints related to the introduction of design patterns. Before introducing a design pattern to a particular design fragment, the basic intent of the pattern should exist in that design fragment already. This starting point is termed a "precursor" in the nomenclature of Ó Cinnéide and Nixon [1], and is not taken into account in much of the existing work in automated refactoring. MORE formulates the notion of precursor as a set of semantic constraints that should be satisfied when introducing design patterns.

The semantic constraint we use for the Factory Method pattern is that the Creator class must create a concrete instance of a Product class [28]. This situation could require the application of the Factory Method pattern, if the developer decides that the Creator class should be able to handle several different types of Product. MORE analyzes, using Soot [37], all the method bodies of a candidate Creator class to retrieve statements containing the operator "new" that occur within its functional methods' body. If the candidate Creator class does not create instances of the Product class, then there is no need to introduce a Factory Method pattern.

The semantic constraints for the Visitor pattern involve the situation when it is required to accumulate new information from an object structure, but the classes of objects in the structure do not support the required behavior [41]. This relates in general to complex hierarchies that have a large number of inherited methods or with God classes that can be detected [10].

The semantic constraints we use for the Singleton pattern is that the class under refactoring (the candidate Singleton): 1) has only one instance, and 2) provide a global point of access to it, i.e., a method called from other classes in the system. These two constraints can be checked using static program analysis technique.

3.3. Multi-objective Formulation of MORE

This section is dedicated to the description of our approach design. We describe how we encoded the optimization problem for the refactoring suggestion using NSGA-II.

Search technique. MORE uses NSGA-II, one of the most popular algorithms that have shown good performance in solving SE problems based on recent surveys. The basic idea of NSGA-II [17] is to make a population of candidate solutions evolve towards the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also called a Pareto set. A non-dominated solution is one where none of the objectives can be improved in value without degrading some of the other objective values.

Solution representation. A candidate solution (individual) is a sequence of refactorings applied to certain elements in the source code under refactoring. We encode a refactoring solution as a vector of length n, where each dimension is a refactoring operation. For each of these refactoring operations, we specify pre-conditions

in the style of Opdyke [3] to ensure that it preserves program behavior as well as and post-conditions to define the effect of the refactoring. Furthermore, for each refactoring operation, specific semantic constraints should be satisfied; otherwise, it is considered as an arbitrary change and therefore it is not considered. The initial population is generated by assigning randomly a sequence of refactorings to some code fragments. To apply a refactoring operation we need to specify which actors, i.e., code fragments, are involved/impacted by this refactoring and which roles they play in performing the refactoring operation. An actor can be a package, class, field, method, parameter, statement or variable.

Solution evaluation. To evaluate the fitness of each refactoring solution, we used three objective functions according to each objective.

- *Anti-patterns objective function:* It calculates the ratio of the number of corrected anti-patterns to the initial number of anti-patterns using the anti-patterns detector component. The anti-patterns correction ratio (ACR) is given by Equation (1):

$$ACR = \frac{number\ of\ corrected\ antipatterns}{initial\ number\ of\ antipatterns} \quad (1)$$

- *Design patterns objective function:* It calculates the number of produced design pattern instances (NP) using the design patterns detector component. NP is given by Equation (2).

$$NP = DPA - DPB \quad (2)$$

when DPA and DPB are the number of design patterns, respectively, after and before refactoring. The NP values are then normalized in the range [0,1] using min-max normalization.

- *Quality objective function:* It calculates the quality improvement. MORE use the QMOOD (Quality Model for Object-Oriented Design) model 26 to estimate the effect of the suggested refactoring solutions on quality attributes. We calculate the overall quality gain (QG) for the six QMOOD quality factors (reusability, flexibility, understandability, effectiveness, functionality, and extendibility) that are formulated using 11 low-level design metrics. For space reasons these metrics are not defined here; full details are available in Bansiya and Davis original work [26]. Let $Q=\{q_1, q_2,... q_6\}$ and $Q'=\{q'_1, q'_2,... q'_6\}$ be respectively the set of quality attribute values before and after applying the suggested refactorings, and $\{w_1, w_2,... w_6\}$ the weights assigned to each of these quality factors. Then the total quality gain (QG) is estimated as follows:

$$QG = \sum_{i=1}^{6} w_i * (q'_i - q_i) \quad (3)$$

Selection and Change operators. To guide the selection process, NSGA-II uses a binary tournament selection based on dominance and crowding distance [17]. NSGA-II sorts the population using the dominance principle, which classifies individual solutions into different dominance levels. Then, to construct a new population, NSGA-II uses a comparison operator based on a calculation of the crowding distance [17] to select potential individuals having the same dominance level. Change operators such as crossover and mutation aim to drive the search towards optimal or near-optimal solutions. For crossover, we use a single, random, cut-point crossover to construct offspring. It starts by selecting and splitting at random two-parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. This operator must ensure that the length limits are respected by eliminating randomly some refactoring operations. For mutation, we use a mutation operator that picks probabilistically one or more refactoring operations from their associated sequence and replaces them by other ones from the initial list of possible refactorings. After applying crossover and mutation,

refactorings that do not satisfy their pre- and post-conditions will be rejected from the current solution. In addition, when more than 15% of the refactorings have to be rejected from a solution due to preconditions non-satisfaction, the change operator should be repeated.

4. Validation

In this section, we present our validation results to evaluate the efficiency of our approach in fixing anti-patterns, introducing design patterns and improving design quality. We start by presenting our research questions. Then we describe how we designed our experiments to answer these research questions.

4.1. Research questions

With this study, we intend to answer the following three research questions:

- ***RQ1*:** To what extent can the proposed approach improve the quality of software systems?
- ***RQ2*:** How does our approach perform compared to existing search-based refactoring approaches?
- ***RQ3*:** Is our approach useful for software engineers in a real-world setting?

4.2. Experimental Settings

We applied our approach to a benchmark composed of four medium and large-size open-source Java projects: Xerces-J, GanttProject, AntApache, and JHotDraw. Xerces-J is a family of software packages for parsing XML. GanttProject is a cross-platform tool for project scheduling. AntApache is a build tool and library specifically conceived for Java applications Finally, JHotDraw is a GUI framework for drawing editors.
Table 1 provides some descriptive statistics about these four programs. We selected these systems for our validation because they came from four different organisations, involved different kinds of software engineering development and had different sizes, ranging from 21 to 240 KLOC with a large number of both design pattern and anti-pattern instances. As we previously note, in these corpora, we considered four different design patterns (*God class*, *Feature Envy*, *Data Class*, and *Spaghetti Code*) and three different design patterns (*Abstract Method Factory*, *Visitor* and *Singleton*).

Table 1. Studied Systems.

Systems	Release	# classes	KLOC	# anti-patterns	# design patterns
Xerces-J	v2.7.0	991	240	81	36
GanttProject	v1.10.2	245	41	49	15
AntApache	v1.8.2	1191	255	92	38
JHotDraw	v 6.1	585	21	24	18

We designed our experiments to answer our research questions. To answer **RQ1**, we conduct a quantitative and qualitative evaluation to evaluate the efficiency of our approach:
Quantitative evaluation. We evaluate the efficiency of our approach for 1) fixing anti-patterns, 2) introducing design patterns, and 3) improving software quality.

- To evaluate the efficiency of our approach in fixing anti-patterns, we calculate the anti-patterns correction ratio (ACR) as given by Equation (1) on our benchmark.
- To evaluate the efficiency of our approach in introducing design patterns, we calculate the number of new design pattern instances (NP) that are introduced as given by Equation (2).

- To evaluate the efficiency of our approach for improving software quality, we calculate the overall quality gain (QG) using the QMOOD (Quality Model for Object-Oriented Design) model [26] as given by Equation (3).

Qualitative evaluation. To evaluate the usefulness of the suggested refactorings, we performed a qualitative evaluation. We manually inspect and assign a correctness/meaningfulness score of 0 or 1 for each refactoring operation according to its coherence with the program semantics. To this end, we define the metric refactoring meaningfulness (RM) that corresponds to the number of meaningful refactoring operations, in terms of semantic coherence, over the total number of refactorings. RM is given by Equation (4).

$$RM = \frac{\#\ meaningful\ refactorings}{\#\ evaluated\ refactorings} \quad (4)$$

To answer **RQ2**, we compared our approach to state-of-the-art approaches in terms of ACR, NP, and the QG. To this end, we compared our approach to Seng et al. [22], Jensen et al. [20], and Kessentini et al. [34]. These approaches are designed each for a specific purpose, i.e., improve quality metrics or fix design patterns. Thus, to make the comparison fair, we apply the suggested refactorings of each approach, and we calculate our evaluation metrics (ACR, NP, QG, and RM)

To answer **RQ3**, we manually evaluated the usefulness of the introduced design patterns in the current software design by assigning a usefulness score in the range [0,5]. We consider a design pattern as useful if its assigned score is ≥3. We define the metric design patterns usefulness (PU) as follow (5).

$$PU = \frac{\#\ useful\ design\ patterns}{\#\ introduced\ design\ patterns} \quad (5)$$

Due to the stochastic nature of the algorithms/approaches we are studying, they can provide different results for the same problem instance from one run to another. To cater for this issue and to make inferential statistical claims, our experimental study is performed based on 31 independent simulation runs for each algorithm/technique studied. The obtained results are statistically analyzed using the Wilcoxon rank sum test [39] with a 95% confidence level ($\alpha = 5\%$). The Wilcoxon rank sum test is applied between NSGA-II and each of the other techniques: Seng et al. [22], Jensen et al. [20], and Kessentini et al. [34]. Our tests show that the obtained results are statistically significant with p-value<0.05.

Furthermore, as for our evaluation we need to select only one solution from the entire Pareto front, with the same manner for all the trial runs, we use a technique similar to the one described in [27]. Equation 6 is used to choose the solution that corresponds of the best compromise between our three objective functions: anti-patterns correction (ACR), design patterns introduction (NP) and quality gain (QG). We normalized all the objective functions values using min-max normalization in the range [0,1]. In this setting, the ideal solution has the best ACR value (equals to 1), the best NP value (equals to 1), and the highest QG value (equals to 1) Hence, we select, from the Pareto front, the nearest solution to the ideal point in terms of Euclidian distance.

$$bestSol = \underset{i=0}{\overset{n-1}{Min}}\left(\sqrt{(1-ACR[i])^2 + (1-NP[i])^2 + +(1-QG[i])^2}\right) \quad (6)$$

where *n* is the number of solutions in the Pareto front returned by NSGA-II.

4.3. Results and discussions

In this section we present the answer to each research question in turn, indicating how the results answer each

question.

Results for RQ1. The results relating in RQ1 are described in Table 2. After applying the proposed refactoring operations by our approach (MORE), we found that, on average, 86% of the detected anti-patterns were fixed (ACR) for all the four studied systems. This high score is considered significant to improve the quality of the refactored systems by fixing the majority of anti-patterns that were from different types (God class, Feature Envy, Data Class, and Spaghetti Code). We found that the majority of non-fixed anti-patterns are related to the *God class* type. This type of anti-patterns usually requires a large number of refactoring operations and is known to be very difficult to correct.

Moreover, we found that MORE succeeded in producing design pattern instances. Table 2 shows the number of new design pattern instances for each system. MORE successfully introduced a median of 7 design patterns (NP) that were from different types (Factory Method, Visitor and Singleton) for all the four studied systems. This can be very helpful for software engineers who might be interested to the introduction of design patterns to make their software systems more understandable, flexible, and maintainable. In addition, when applying the suggested refactorings we noticed that some God classes are fixed when involved in introducing a visitor pattern. For instance, we observe that the God class `GanttTree` in GanttProject, was fixed automatically when introducing a visitor pattern. In addition, the new structure of this class become more flexible with the visitor pattern where new functionalities and behavior can be easily added without affecting the original class.

Table 2. ACR, NP, and QG median values of 31 independent runs of MORE, Seng et al., Jensen et al., and Kessentini et al. The p-values of the Wilcoxon rank sum test indicate whether the median of the approach (Seng/Jensen/Kessentini) is statistically different from MORE with a 95% confidence level ($\alpha = 0.05$). A statistical difference is accepted at $p<=0.05$.

Systems	**Algorithms**	**ACR**		**NP**		**QG**	
		Score	*p-value*	*Score*	*p-value*	*Score*	*p-value*
Xerces-J	MORE	89%		12		0.47	
	Seng et al.	23%	0.05	0	0.01	0.54	0.02
	Jensen et al.	14%	0.04	31	0.01	0.41	0.01
	Kessentini et al.	88%	0.04	0	0.01	0.32	0.01
GanttProject	MORE	88%		7		0.34	
	Seng et al.	24%	0.02	1	0.01	0.33	0.01
	Jensen et al.	33%	0.05	14	0.01	0.35	0.01
	Kessentini et al.	84%	0.05	0	0.01	0.21	0.01
AntApache	MORE	86%		4		0.5	
	Seng et al.	7%	0.04	0	0.01	0.52	0.01
	Jensen et al.	12%	< 0.01	28	< 0.02	0.51	< 0.01
	Kessentini et al.	87%	< 0.01	0	< 0.01	0.39	< 0.01
JHotDraw	MORE	83%		4		0.17	
	Seng et al.	38%	< 0.01	0	< 0.01	0.19	< 0.01
	Jensen et al.	25%	< 0.01	9	< 0.01	0.14	< 0.01
	Kessentini et al.	88%	< 0.01	0	< 0.01	0.1	< 0.01
Average (all systems)	MORE	86%		7		0.37	
	Seng et al.	23%		0.25		0.39	
	Jensen et al.	21%		20.5		0.35	
	Kessentini et al.	86%		0		0.25	

In terms of quality improvement (QG), as can be seen in Table 2, MORE succeeded in improving the quality of the four studied systems, with an average QG score of 0.37 in terms of QMOOD quality attributes. In figure 2, we show the obtained QG values that we calculated for each QMOOD quality attribute before and after refactoring for each studied system. We found that the systems quality increase across the four QMOOD quality factors. Understandability is the quality factor that has the highest QG value; whereas the effectiveness quality factor has the lowest one. This due to two possible reasons 1) the majority of non-fixed anti-patterns (God class, spaghetti code) are known to increase the coupling (DCC) within classes which heavily affect the quality index

calculation of the Effectiveness factor; 2) the vast majority of suggested refactoring types were move method, move field, and extract class that are known to have a high impact on coupling (DCC), cohesion (CAM) and the design size in classes (DSC) that serves to calculate the understandability quality factor. Furthermore, we noticed that JHotDraw produced the lowest quality increase for the four quality factors. This is justified by the fact that JHotDraw is known to be of good design and implementation practices [10] and it contains a few anti-patterns comparing to the three other studied systems.

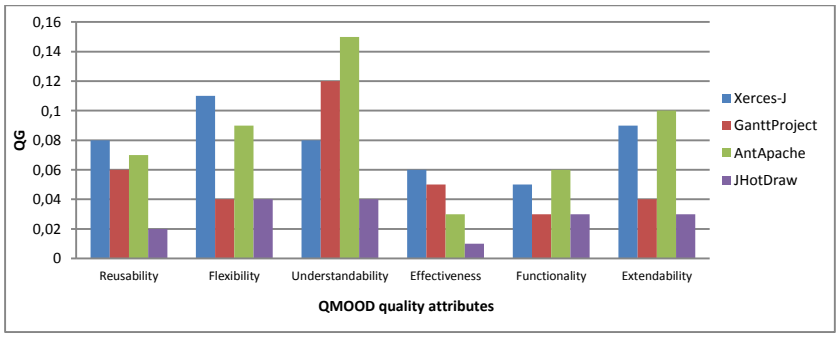

Fig 2. QMOOD quality factors gain obtained by MORE

The obtained results are promising, however, improving the design structure is not always enough to determine whether our approach produce a coherent program and fit with software engineers expectations. Figure 3, describes the results of our qualitative evaluation. We found that the majority of the suggested refactorings (an average of 86% over the four studied systems) could be successfully applied to the program and only a small number of the suggested refactorings were rejected due to semantic incoherencies in the source code.

To sum up, we can conclude that our approach succeeded in improving the code quality not only by fixing the majority of detected anti-patterns and introducing a considerable number of design patterns but also by a significant improvement on the overall design quality of the refactored program such as the user understandability, the reusability, and the flexibility. At the same time, the proposed refactoring operations are considered as semantically feasible and do not affect the semantic coherence of the refactored program from the point of view of potential users.

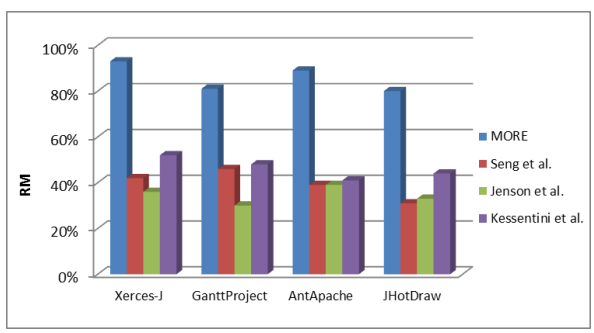

Fig. 3. Refactoring meaningfulness (RM) evaluation.

Results for RQ2. The results relating to RQ2 are summarized in Table 2. As described in Table 2, after applying the proposed refactoring operations, we found that more than 86% of detected anti-patterns were fixed (ACR) as an average for

all the four studied systems. For instance, for GanttProject, 75% (9 over 12) of God classes, 86% (6 over 7) of feature envy, 94% (15 over 16) of spaghetti code, 93% (13 over 14) of Data classes are fixed. This score is comparable to the correction score of Kessentini et al having an average of 86%. However, the obtained results are much better than those of Seng et al., and Jensen et al. having respectively only 23% and 21%, on average for all the studied systems.

In terms of patterns introduction, Jensen et al. produces the higher score by introducing, on average for the four systems, 20.5 design patterns. This score is higher than the one obtained by MORE (an average of 7 patterns per system). This can be explained by the fact that Jensen et al. apply design patterns without considering if the design pattern is needed or not in that code fragment, i.e., the sole aim is to produce more design patterns. This is unlikely to be useful and efficient in practice. For Seng et al. and Kessentini et al. we found that they are not able to produce design patterns. This is because the lists of refactorings they use are not geared for the introduction of design patterns.

Furthermore, MORE produces comparable QG values to Seng et al. and Jensen et al. having respectively 0.37, 0.39 and 0.35, since the quality metrics improvement is a common component in the objective function of each approach. However Kessentini et al. produce a lower QG score since their approach is driven only by anti-pattern correction and not by improving quality metrics. On the other hand, despite the significant improvement in terms of QG for Seng et al. (the highest score), it is not effective at fixing anti-patterns (only 23% of anti-patterns are fixed). Thus these results provide evidence to support the claim that improving quality metrics does not necessarily mean that existing anti-patterns are fixed.

More notably, we compared MORE to the three other approaches in terms of semantic coherence. Figure 3 summarizes our findings. Regarding the refactoring meaningfulness, for all of our four studied systems, an average of 86% of proposed refactoring operations are considered as semantically feasible and do not generate semantic incoherence. This score is significantly higher than the scores of the three other approaches having respectively only 40%, 35% and 46%, as RM scores for respectively, Seng et al., Jensen et al., and Kessentini et al. Thus, our approach performs clearly better for RM. Moreover, we noticed that for the larger programs, the performance in terms of refactoring meaningfulness (RM) achieved by MORE is more notable than it is for the smaller programs.

Results for RQ3. The results relating to RQ3 are summarized in Table 3. We observe that the majority (more than 83%) of the design patterns produced by MORE are considered as useful in the four studied systems since their introduction is guided by a set of semantic constraints and not arbitrary. However, we found that a relatively small number of patterns produced by Jensen et al. (less than 36%) are considered as feasible. The main reason is that these design patterns are applied in an arbitrary way, without considering if they are needed in that code fragment or not.

Thus MORE produces higher increases in RM than the other three approaches, which is probably the cause of the significant score in terms of patterns usefulness.

Table 3. Comparison of MORE with Jensen et al. in terms of Patterns usefulness (PU)

Systems	MORE	Jensen et al.
Xerces-J	83% (10|12)	35% (11|31)
GanttProject	86% (6|7)	36% (5|14)
AntApache	100% (4|4)	14% (4|28)
JHotDraw	100% (4|4)	22% (2|9)

Furthermore, it is important to evaluate the scalability of our approach. Indeed, there is a pressing need for scalable solutions to Software Engineering problems. Scalability is widely regarded as one of the key problems for Software Engineering research and development [47]. To evaluate the scalability of the performance of our approach for systems of increasing size, we executed MORE on the four studied systems that were from different sizes ranging from 245 for JHotDraw to 1191 classes for AntApache. As shown in Figure 4, when the

size of the systems increase, the execution time do not significantly increase in turn.

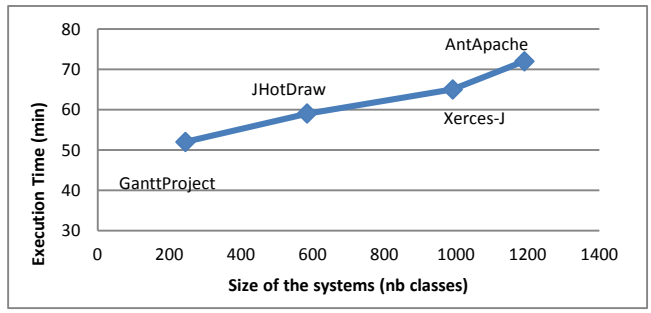

Fig. 4. Scalability of MORE on different systems.

Despite these encouraging results, our approach presents some limitations that should be considered. First, some of design pattern instances are hard to achieve and require an extra manual effort from the developer to tailor the implementation to fit the context. Second, although our approach succeeded in introducing the Factory Method, Visitor, and Singleton design patterns, we cannot generalize the results for other design pattern types. In addition, some refactoring solutions require a significant number of code changes which may take the code away from its initial design. To address this issue, we plan to consider new criteria to reduce the amount of code changes when recommending refactoring. Furthermore, in large-scale systems, the number of anti-patterns to fix can be very large and not all of them can be fixed automatically. Thus, the prioritization of the list of anti-patterns is required based on different criteria such as the severity, and the risk.

5. Conclusion

We propose, in this paper, an automated multi-objective refactoring recommendation approach to improve design quality (as defined by software quality metrics), fix anti-patterns, and introduce design patterns. To evaluate our approach, we conducted a quantitative and qualitative evaluation using a benchmark composed of four open source systems. The statistical analysis of the results confirms the efficiency of our proposal compared to the state-of-the-art of refactoring techniques.

As part of our future work, we are planning to extend the validation of our work to evaluate our proposal with additional types of design patterns and anti-patterns, and prioritize the correction of anti-patterns. In addition, we plan to perform an empirical study to evaluate the correlation between introducing design patterns and their impact on several types of anti-patterns.

Acknowledgments. This work is supported by “Collecting, Analyzing, and Evaluating Software Assets for Effective Reuse”, Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003), and “Software License Evolution Analysis”, Osaka University Program for Promoting International Joint Research.

References

1. M. Ó Cinnéide, Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College, 2001.
2. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts: Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, 1999

3. W. F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.
4. S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, A comparative study of manual and automated refactorings, in 27th European Conference on Object-Oriented Programming (ECOOP), 2013, pp. 552–576.
5. M. Kim, T. Zimmermann, and N. Nagappan, A field study of refactoring challenges and benefits, in 20th International Symposium on the Foundations of Software Engineering (FSE), 2012, pp. 50:1–50:11.
6. E. R. Murphy-Hill, C. Parnin, and A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 5–18, 2012.
7. E. R. Murphy-Hill and A. P. Black, Refactoring tools: Fitness for purpose, IEEE Software, vol. 25, no. 5, pp. 38–44, 2008.
8. Xi Ge, E. R. Murphy-Hill, BeneFactor: a flexible refactoring tool for eclipse. OOPSLA Companion 2011: 19-20
9. D. Silva, R. Terra, M. T. Valente, Recommending Automated Extract Method Refactorings, Internation Conference on Program Comprehension ICPC 2014.
10. A. Ouni, M. Kessentini, H. Sahraoui and M. Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach. J. of Autmated Software Engineering, Springer, 2012.
11. M. Harman, and L. Tratt, Pareto optimal search based refactoring at the design level, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07), pp. 1106-1113, 2007.
12. M. O'Keeffe, and M. O. Cinnéide, Search-based Refactoring for Software Maintenance. J. of Systems and Software, 81(4), 502–516, 2008.
13. T. Mens, T. Tourwé: A Survey of Software Refactoring. IEEE Trans. Software Eng. 30(2), pp. 126-139, 2004.
14. W. F. Opdyke, R. E. Johnson, Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), September 1990.
15. W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, March 1998
16. Xi Ge and Emerson Murphy-Hill. Manual Refactoring Changes with Automated Refactoring Validation. In Proceedings of the Int. Conf. on Soft. Eng. (ICSE), 2014.
17. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput., vol. 6, pp. 182–197, Apr. 2002.
18. A. Ajouli, J. Cohen, J. Royer, Transformations between Composite and Visitor Implementations in Java. EUROMICRO-SEAA, pp. 25-32, 2013
19. M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, JDeodorant: identification and application of extract class refactorings. International Conference on Software Engineering (ICSE), pp. 1037-1039, 2011.
20. A. Jensen and B. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Proceedings of GECCO. ACM, July 2010.
21. F. Qayum, R. Heckel, Local search-based refactoring as graph transformation. Proceedings of 1st International Symposium on Search Based Software Engineering; pp. 43–46, 2009.
22. O. Seng, J. Stammel, and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In GECCO'06, pp. 1909–1916, 2006.
23. F. Steimann and A. Thies. From behaviour preservation to behaviour modification: constraint-based mutant generation. In Int. Conf. on Software Engineering (ICSE), pp. 425–434, 2010.
24. G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba. Supporting Extract Class Refactoring in Eclipse: The ARIES Project. In Proceedings of the 34th Int. Conf. on Software Engineering (ICSE), pp. 1419-1422, 2012.
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Reading, MA, 1995.

26. J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Engg., 28(1): 4–17, 2002.
27. A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi: Search-based refactoring: Towards semantics preservation. ICSM 2012: 347-356
28. Mel Ó Cinnéide, Paddy Nixon: A Methodology for the Automated Introduction of Design Patterns. ICSM 1999.
29. M.V. Mäntylä, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, in: IEEE International Conference on Software Maintenance (ICSM), pp. 381–384, 2003.
30. W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, Journal of Systems and Software 80 (2007) 1120–1128.
31. Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 61 pages.
32. D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In SCAM 04, pages 65–74, 2004.
33. M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, Experimental Assessment of Software Metrics Using Automated Refactoring, Proc. Empirical Software Engineering and Management (ESEM), pages 49-58, 2012.
34. M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design Defects Detection and Correction by Example, 19th IEEE ICPC11, pp. 81-90, Canada, 2011.
35. A. Ouni, M. Kessentini, H. Sahraoui, M. Hamdi: The use of development history in software refactoring using a multi-objective evolutionary algorithm. GECCO 2013
36. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible? in Int. Conf. on Compiler Construction, pp. 18–34, 2000.
37. P. Lam, E. Bodden, O. Lhotak and L. Hendren, The Soot framework for Java program analysis: a retrospective, Cetus Users and Compiler Infrastructure Workshop, October 2011.
38. D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design Pattern Detection, International Workshop on Program Comprehension (IWPC) 2003
39. Arcuri, A. and Briand, L. C. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), 2011.
40. A. Ajouli, Vues et Transformations de Programmes pour la Modularité des Évolutions, Ph.D. dissertation, University of Nantes Angers Le Mans, 2013.
41. Joshua Kerievsky. Refactoring to Patterns. Pearson Higher Education, 2004.
42. A. Arcuri and G. Fraser, Parameter tuning or default values? An empirical investigation in search-based software engineering, Empirical Software Engineering 18(3), Springer, 2013.
43. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Journal of software maintenance and evolution: Research and practice. J. Softw. Maint. Evol., 17(4):247–276, 2005.
44. G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. Int. conference on Automated software engineering, pages 151–154, 2010.
45. Emerson R. Murphy-Hill, Andrew P. Black: Programmer-Friendly Refactoring Errors. IEEE Trans. Software Eng. 38(6): 1417-1431 (2012)
46. Emerson R. Murphy-Hill, Andrew P. Black: Breaking the barriers to successful refactoring: observations and tools for extract method. ICSE 2008: 421-430.
47. I. Sommerville, Software Engineering, 6th ed. Addison-Wesley, 2001.
48. Ali Ouni, Marouane Kessentini, Houari A. Sahraoui: Multiobjective Optimization for Software Refactoring and Evolution. Advances in Computers 94: 103-167 (2014)